2

RSRE
MEMORANDUM No. 4195

ROYAL SIGNALS & RADAR ESTABLISHMENT

SPECIFICATION OF VIPER1 IN Z

Author: D H Kemp

PROCUREMENT EXECUTIVE,
MINISTRY OF DEFENCE,
R S R E MALVERN,
WORCS.

ROYAL SIGNALS AND RADAR ESTABLISHMENT
Memorandum 4195

Title

Specification of Viper1 in Z

Author

D.H. Kemp

Date

September 1988

Summary

The Viper1 microprocessor has already been specified mathematically in HOL. HOL, however, is not well known outside the hardware verification community. This paper covers the specification Viper1 in the Z specification language. Various features of Viper1 have been specified in Z which did not occur in the top level HOL specification. It has not been possable to prove any correspondence between this specification and the original HOL specification. The Work involved in writing the Viper1 specification has proved useful in writing the initial Viper2 specification.

1 Introduction

In safety critical systems the idea of diversity improving safety is well established. Safety critical systems may employ a number of processors independently executing algorithms which obey a common high level specification.

The same can be seen to be true in the specification and design of safety critical systems. Just as there may be an error in a specific microprocessor, which could cause one channel of a system to fail in service, there may be an error in a specification. The chances of this escaping the notice of the designers is reduced by specifying the system in more than one way. This is most effective when the two specification systems are basically different in character. This means then that by ensuring the system conforms to both of the specifications the chance of an error still being present is greatly reduced. Proofs of correspondence may then be attempted to establish that the two texts have the same meaning.

It was decided to specify Viper1 in Z for a number of reasons. Firstly Z was sufficiently different to the HOL specification[1] to give reasonable diversity. Z is also more widely known in terms of software specification than HOL. Another advantage of Z was the fact that it had been used by J.Bowen[2] to specify the M6800 microprocessor. A lot of the groundwork developed by Bowen in specifying a microprocessor instruction set has been used in this specification. Finally at RSRE there is a Z editor and type checker available for use on the PerqFlex system.

This report is the first attempt to specify Viper1 in Z. It makes no attempt to explain the primary constructs of Z, nor act as a tutorial in Z. Readers not familiar with Z should consult reference 3. Although the specification has been type checked, it has neither been proved to be equivelent to the HOL specification, nor free from logical errors. Any inconsistencies or errors found in this document should be reported back to the Computing Division at RSRE.

2 Basic functions

2.1 Bits and Words

Initially the models adopted to represent bits and words need to be defined, along with the relationships between these models and the natural numbers which they represent.

```
Bit ≙ {0,1}

Word ≙ { w:ℕ⇸Bit | #w>0 ∧ dom w = 0 .. (#(w) - 1) }
```

Bits are represented as the set of elements with values 0 or 1. Words are represented as a set of partial functions from natural numbers to Bits. The natural numbers correspond to the position of the bit in the word, ie the result of w(n) (the word w acting on the value n) gives the n+1th Bit of the word w.

```
| LSB,MSB : Word → Bit
|——————————————————————
| ∀ w : Word •
| LSB w = w 0
| MSB w = w #w-1
```

Find the most and least significant bits of the word.

```
| val : Word → ℕ
|————————————————————————————————————————————
| ∀ w : Word •
|    (#w=1 ) ⟹ ( val w = LSB w  )
|    (#w>1 ) ⟹ ( val w = LSB w + 2 * val(succ⨾w))
```

val returns the natural number represented by the word.Note succ⨾w gives the effect of a Right shift, ie divide by two, on the word. ie if succ⨾w is applied to n then first succ n is calculated, and then w of n+1 is calculated ie the n+2th Bit is returned rather than the n+1th one.

```
| pred : ℕ₁ → ℕ
|——————————————————————————
| ∀ n : ℕ • pred n = n - 1
```

Useful for left shifting (in a similar way to the technique described above).

```
| (_set_) : (Word×Bit) → Word
|——————————————————————————————
| ∀ w : Word; b : Bit •
|   w set b = w⊕{(0↦b),(1↦b)}
```

The set function returns a word which has all of its bits set to the specified value.

$$\begin{array}{|l} maxval : Word \to \mathbb{N} \\ \hline \forall\ w : Word \bullet \\ \quad maxval\ w = val(w\ set\ 1) \end{array}$$

$$\vdash \neg\ (\ \exists\ w : Word \bullet ((\ val\ w) > maxval\ w)\ )$$

Returns the maximum value which can be stored in the word.

$$\begin{array}{|l} wrd : \mathbb{N}_1 \to (\ \mathbb{N} \to Word\ ) \\ \hline \forall\ size : \mathbb{N}_1;\ valu : \mathbb{N};\ w : Word \bullet \\ \quad (wrd\ size\ valu = w\ ) \Leftrightarrow \\ \qquad ((\#w = size) \land \\ \qquad\ (val\ w = valu\ mod\ succ(maxval\ w))) \end{array}$$

wrd returns the word of size size and set to the value valu (unless the word cannot hold that value). (note no algorithm is given for calculating wrd from its arguments, just the relationships which must hold between the word returned and the input arguments).

$$\begin{array}{|l} (\_\hat{}\_) : (\ Word \times Word) \to Word \\ \hline \forall\ w1,w2 : Word \bullet \\ \quad w1\hat{}w2 = w1 \quad \cup \quad (\ pred^{\#w1}\ ;\ w2\ ) \end{array}$$

$$\vdash \forall\ w1,w2 : Word \bullet \#(w1\hat{}w2) = \#w1 + \#w2$$

Concatenate two words together.

2.2 Bitwise Functions

The definition of the basic logical functions.

$$\begin{array}{|l} \mathrm{not} : \mathrm{Bit} \rightarrowtail \mathrm{Bit} \\ \hline \mathrm{not} = \{0 \mapsto 1, 1 \mapsto 0\} \end{array}$$

Generate the logical inverse of the input bit.

$$\begin{array}{|l} (\_\cdot\_), (\_+\_), (\_\circ\_) : (\mathrm{Bit} \times \mathrm{Bit}) \rightarrow \mathrm{Bit} \\ \hline (\_+\_) = \{(0,0) \mapsto 0, (0,1) \mapsto 1, (1,0) \mapsto 1, (1,1) \mapsto 1\} \\ (\_\circ\_) = \{(0,0) \mapsto 0, (0,1) \mapsto 1, (1,0) \mapsto 1, (1,1) \mapsto 0\} \\ (\_\cdot\_) = \{(0,0) \mapsto 0, (0,1) \mapsto 0, (1,0) \mapsto 0, (1,1) \mapsto 1\} \end{array}$$

Standard bitwise logical functions. (note, $\cdot$ is, $+$ is or and $\circ$ is exclusive or)

2.3 Logical fuctions on words

The standard wordwise logical functions, ie finding the logical AND of two words.

```
│ wnot : Word ⤖ Word
├──────────────────────
│ ∀ w : Word •
│   wnot w = w ⨾ not
```

Generate the inverse of the input word.

```
WordPair ≙
    { w : ℕ ⇸ (Bit×Bit) | #w>0 ∧ dom w = 0 .. ((#w)-1) }
```

```
│ (_pair_) : (Word×Word)→WordPair
├──────────────────────────────────────────────────────
│ ∀ w1,w2 : Word •
│   w1 pair w2 =
│     { i : ℕ | i ∈ dom w1 ∩ dom w2 • i ↦ ( w1 i,w2 i) }
```

Takes a pair of words and represents them as a set of bit pairs, indexed by a single natural number.

```
│ (_and_),(_or_),(_exor_) : (Word×Word)→Word
├──────────────────────────────────────────────
│ ∀ w1,w2 : Word •
│ w1 and  w2 = ((w1 pair w2 ) ⨾ (_·_))
│ w1 or   w2 = ((w1 pair w2 ) ⨾ (_+_))
│ w1 exor w2 = ((w1 pair w2 ) ⨾ (_⊕_))
```

Standard wordwise logical functions.

```
│ (_<<_) : (Word×Bit) → Word
├──────────────────────────────────────────
│ ∀ w : Word; b : Bit •
│   w << b = ( {#w} ⩤ (pred ⨾ w) ) ∪ {0↦b}
```

```
│ (_>>_) : (Bit×Word) → Word
├──────────────────────────────────────────
│ ∀ w : Word; b : Bit •
│   b >> w = {((#w)-1) ↦ b} ∪ (succ ⨾ w )
```

Shift right and left while inserting a particular bit into the right or left most position.

2.4 Arithmetic Functions

Next the mathematical functions must be defined. This includes introducing integers (ie 2's complement notation), and standard mathematical operations and exceptions (for example add and carry).

$$\begin{array}{|l}
\text{value} : \text{Word} \rightarrow \mathbb{Z} \\
\hline
\forall\, w : \text{Word} \bullet \\
\quad ((\text{MSB}\ w = 1) \wedge \text{value}\ w = \text{val}\ w - \text{succ}\ (\text{maxval}\ w\ )) \vee \\
\quad ((\text{MSB}\ w = 0) \wedge \text{value}\ w = \text{val}\ w\ )
\end{array}$$

Return the integer value represented by the Word.This is using the 2's complement notation. The most significant bit has a weighting of -2^{n-1}. So to cope with negative numbers subtract 2^n.

$$\begin{array}{|l}
\text{maxpos}, \text{maxneg} : \text{Word} \rightarrow \mathbb{Z} \\
\hline
\forall\, w1, w2 : \text{Word} \mid \#w1 < 1 \wedge \#w2 = ((\#w1)-1) \bullet \\
\qquad \text{maxpos}\ w1 = \text{maxval}\ w2 \\
\qquad \text{maxneg}\ w1 = (\ \text{maxval}\ w2) - (\ \text{maxval}\ w1)
\end{array}$$

Return the maximum positive and negative numbers for a word of a particular size.

$$\begin{array}{|l}
(\_\text{signextend}\_) : (\text{Word} \times \mathbb{N}_1) \rightarrow \text{Word} \\
\hline
\forall\, w1, w2 : \text{Word};\ \text{length} : \mathbb{N}_1 \mid \\
\qquad\qquad\qquad\qquad (\text{length} \geq \#w1) \wedge (\#w2 = \text{length}) \bullet \\
\quad (w1\ \text{signextend}\ \text{length}) = (w2\ \text{set}\ (\text{MSB}\ w1)) \bullet w1
\end{array}$$

Sign Extend the word to the new word length.

$$\begin{array}{|l}
(\_\text{pad}\_) : (\text{Word} \times \mathbb{N}_1) \rightarrow \text{Word} \\
\hline
\forall\, w1, w2 : \text{Word};\ \text{length} : \mathbb{N}_1 \mid \\
\qquad\qquad\qquad\qquad (\text{length} \geq \#w1) \wedge (\#w2 = \text{length}) \bullet \\
\quad (w1\ \text{pad}\ \text{length}) = (w2\ \text{set}\ 0) \bullet w1
\end{array}$$

Pad out a word to the new word length with zeros.

$$\begin{array}{|l}
(\_\text{trim}\_) : (\text{Word} \times \mathbb{N}_1) \rightarrow \text{Word} \\
\hline
\forall\, w : \text{Word};\ \text{length} : \mathbb{N}_1 \mid \text{length} \leq \#w \bullet \\
\quad w\ \text{trim}\ \text{length} = (0\ ..\ \text{length}) \lhd w
\end{array}$$

Trim a word down to the new word length.

```
(_plus_) : (Word×Word) → Word
──────────────────────────────────────────────────────────
∀ w1,w2,w3 : Word | (#w1) = ((#w2)+1) ∧ (#w2) =(#w3)  •

  ((w2 plus w3) = (w1 trim #w2))
                 ⇔ (value w1)=(value w2)+(value w3)
```

Primitive addition. All that is checked for is that the input words are of the same size, and that the output word is one bit larger, so that carry can be detected. Word addition is defined in terms of integer addition, ie addition per se is not defined.

```
(_minus_) : (Word×Word) → Word
──────────────────────────────────────────────────────────
∀ w1,w2,w3 : Word | (#w1) = ((#w2)+1) ∧ (#w2) =(#w3'  •

  ((w2 minus w3) = (w1 trim #w2))
                 ⇔ (value w1)=(value w2)-(value w3)
```

Subtraction is defined similarily to addition, note no checks for overflow etc.

```
(_carry_) : (Word×Word) → Bit
───────────────────────────────────────────────────────────
∀ w1,w2 : Word | #w1 = #w2 •
  (w1 carry w2 = 1) ⇔ ((val w1) + (val w2) > maxval w1)
```

Top level specification of carry, ie a carry is generated when the addition result is larger than the maximum possible value which can be stored.

```
(_borrow_) : (Word×Word) → Bit
──────────────────────────────────────────────────
∀ w1,w2 : Word | #w1 = #w2 •
  (w1 borrow w2 = 1) ⇔ ( (val w1) < (val  w2) )
```

Top level spec of Borrow.

```
(_overflow_) : (Word×Word) → Bit
───────────────────────────────────────────────────────
∀ w1,w2 : Word | #w1 = #w2 •
  (w1 overflow w2 = 1) ⇔
        ( ( (value w1) + (value w2) > maxpos w1  ) ∨
          ( (value w1) + (value w2) < maxneg w2  ) )
```

Top level spec of overflow, ie overflow when the sum is greater than the largest positive value which can be held, or less than the largest negative number.

```
| (_underflow_) : (Word×Word) → Bit
|--------------------------------------------------
| ∀ w1,w2 : Word | #w1 = #w2 •
|   (w1 underflow w2 = 1) ⇔
|         ( ( (value w1) - (value w2) > maxpos w1  ) ∨
|           ( (value w1) - (value w2) < maxneg w2  ) )
```

Top level spec of overflow on subtraction

```
| (_equal_) : (Word×Word) → Bit
|----------------------------------------------
| ∀ w1,w2 : Word | #w1 = #w2 •
|   (w1 equal w2 = 1) ⇔ (value w1 = value w2)
```

Set to 1 if the two words have the same value (and 0 otherwise), note they are not necessarily the same size of word.

```
| (_less_) : (Word×Word) → Bit
|----------------------------------------------
| ∀ w1,w2 : Word | #w1 = #w2 •
|   (w1 less w2 = 1 ) ⇔ ( value w1 < value w2)
```

Set to 1 if the first word is less than the second (and 0 otherwise), note they are not necessarily the same size of word.

This completes the underlying theory of representing natural number arithmetic by operations on vectors of bits.

3 Viper Specifics

3.1 Word Lengths

These are the specific word sizes used in the Viper1 processor.

$Word_{32} \triangleq \{ w : Word \mid \#w = 32 \}$
-- for Data words

$Word_{20} \triangleq \{ w : Word \mid \#w = 20 \}$
-- For Address words

$Word_{4} \triangleq \{ w : Word \mid \#w = 4 \}$
-- For the function select

$Word_{3} \triangleq \{ w : Word \mid \#w = 3 \}$
-- for the destination select

$Word_{2} \triangleq \{ w : Word \mid \#w = 2 \}$
-- for the register and memory select

$Word_{1} \triangleq \{ w : Word \mid \#w = 1 \}$
-- for the comparison select and flags

$Address \triangleq Word_{20}$

$Data \triangleq Word_{32}$

$Flag \triangleq Word_{1}$

3.2 Memory

The definition of the Memory and Peripheral spaces, and the behaviour of these two regions.

```
┌─Memory──────────────────────────────
│ Mem       : Address   → Data
│ RAMspace  : Address   → Data
│ PERIspace : Address   → Data
│ io        : Bit
├─────────────────────────────────────
│ (io = 0) ⇒ (Mem = RAMspace)
│ (io = 1) ⇒ (Mem = PERIspace)
└─────────────────────────────────────
```

Two regions of non overlapping address space RAM and PERIpheral. The two types of memory totally cover the memory space.

```
┌─ΔMemory─────────────────────────────
│ Memory
│ Memory'
│ δMem : Address ⇸ Data
├─────────────────────────────────────
│ (io = 0) ⇒ (Mem' = Mem ● δMem)
└─────────────────────────────────────
```

If the location is in RAM then the address is updated, however with PERIpheral space the values can change without any modification from the processor. No mention of the behaviour of the PERIpheral space is given, because there is no way to model in general these very specific devices. The specifiction of the behaviour of these devices is left to the system specification.

```
┌─ΞMemory──────
│ ΔMemory
├──────────────
│ δMem = ∅
└──────────────
```

No change in memory.

3.3 Registers

The specification of the Viper1 registers.

Registers

$$\begin{array}{|l} A : Word_{32} \\ X : Word_{32} \\ Y : Word_{32} \\ P : Word_{20} \\ B : Word_{1} \\ \hline \end{array}$$

The five visible registers of the Viper. A an accumulator, X and Y index registers P the program counter and B the boolean flag.

ΔRegisters

$$\begin{array}{|l} Registers \\ Registers' \\ newp \;\; : Address \\ \hline \end{array}$$

Note, P' is always updated (unless machine has stopped).

ΞRegisters

$$\begin{array}{|l} \Delta Registers \\ \hline A' = A \\ X' = X \\ Y' = Y \\ B' = B \\ \hline \end{array}$$

ie no change.

3.4 Clock

The existance of a clock was not represented in any manner in the HOL specification of Viper1, but it is included here as a matter of completeness.

```
┌─ Clock ───
│ Clk : ℕ
└──────────
```

Clock simply counts up from 0.

```
┌─ ΔClock ──────────
│ Clock
│ Clock'
│ Cycles : ℕ
├─────────────────
│ Clk' = Clk + Cycles
└──────────────────
```

Cycles is the number of cycles needed to complete the present instruction. It is intended to include information about how many cycles each instruction takes to complete in the schemas of the individual instructions.

3.5 Stop

The definition of the stop flag and the way the processor behaves when stopped and in the normal mode of operation.

```
┌─Stop──────
│ stop : Bit
└───────────
```

Single Bit top determine whether the machine is stopped or not.

```
┌─ΔStop─────────────────────
│ ΔRegisters
│ Stop
│ Stop'
│ sval  : Bit
│ reset : Bit
├───────────────────────────
│ stop  = 0
│ reset = 0
│ stop' = sval
│ newp  = P plus (wrd 20 1)
└───────────────────────────
```

Set the new value of the program counter and the stop bit for the next state. The machine is not stopped. The parameter reset is the reset line to the processor. It is treated as a synchronous reset, ie *it is only noticed at the start of an instruction.*

```
┌─Stopped──────
│ ΞMemory
│ ΞRegisters
│ Stop
│ Stop'
│ ΔClock
│ reset : Bit
├──────────────
│ stop  = 1
│ reset = 0
│ P' = P
└──────────────
```

The machine has stopped, and cannot restart until there is a Reset.

3.6 Viper State

ViperOpCode

op : $Word_{32}$
rsf : $Word_2$
msf : $Word_2$
dsf : $Word_3$
csf : $Word_1$
fsf : $Word_4$
addr : $Word_{20}$

op = rsf ^ msf ^ dsf ^ csf ^ fsf ^ addr

The Viper1 Op code.The Op code is loaded in from the location pointed to by the Program counter (P). The Op code constists of six fields. These are,

(1) The Register Select Field - This selects which of the four registers are going to be used as inputs to the ALU.

(2) The Memory Select Field - This selects the addressing mode for the operation.

(3) Destination Select Field - This selects the destination register for the result, and also whether the result is a Jump or a Call.

(4) Comparison Select Field - This selects whether the operation is a comparison (setting the B flag) or an arithmetic or logical function, returning a result. It is also used to distinguish between Jump and Call instructions.

(5) The Function Select Field - This determins the ALU operation for Comparisons or Arithmetic and Logical functions.

(6) The Address Field - The address used to pull in the second operand from memory, or used as Jump address etc.

ArithmeticAndLogicalUnit

result : $Word_{32}$
offs : $Word_{32}$
r,m : $Word_{32}$

The inputs and outputs to/from the ALU. r holds the value from the register, specified in the register select field of the Op code, which is the first operand to the ALU. The parameter offs is the address of the memory input to the ALU (or the actual input if the operation is in immediate addressing mode). The parameter m is the actual value passed as the second operand to the ALU.Finally result is the output from the ALU.

```
┌─ΔViper ──────────────────────
│ ΔMemory
│ ΔRegisters
│ ΔClock
│ ΔStop
│ ViperOpCode
│ ArithmeticAndLogicalUnit
│ bval    : Bit
├─────────────────────────────
│ op = Mem (P)
└─────────────────────────────
```

The Viper State. For the machine to change to a new state then the machine must not be stopped.

```
┌─ΞViper ─────
│ ΔViper
│ ΞMemory
│ ΞRegisters
│ ΔStop
└─────────────
```

Viper state unchanged (exept P updated)

```
┌─ViperINIT ─────
│ ΔViper
├────────────────
│ Clk' = 0
│ stop' = 0
│ val (P' ) = 0
│ val (A' ) = 0
│ val (X' ) = 0
│ val (Y' ) = 0
│ val (B' ) = 0
└────────────────
```

Machine on start up.

```
Reset
ΔMemory
ΔRegisters
ΔClock
Stop'
reset : Bit
──────────────
reset = 1
stop' = 0
val (P') = 0
val (A') = 0
val (X') = 0
val (Y') = 0
val (B') = 0
```

Machine status on a Reset. This was not represented in the HOL specification

4 Viper Operations

4.1 ALU inputs

In this section the various inputs to the Viper1 ALU are specified.

```
┌─ RegisterSelect ──────────────────
│ ΔViper
├──────────────────────────────────
│ (val rsf = 0) ⇒ (r = A)
│ (val rsf = 1) ⇒ (r = X)
│ (val rsf = 2) ⇒ (r = Y)
│ (val rsf = 3) ⇒ (r = P pad 32)
└──────────────────────────────────
```

Select the register to be the r input to the ALU.

```
┌─ Offset ───────────────────────────────────────
│ ΔViper
├────────────────────────────────────────────────
│ (val msf = 0) ⇒ (offs = addr pad 32)
│ (val msf = 1) ⇒ (offs = addr pad 32)
│ (val msf = 2) ⇒ (offs = (addr pad 32) plus X)
│ (val msf = 3) ⇒ (offs = (addr pad 32) plus Y)
└────────────────────────────────────────────────
```

Determine the address of second word to be input to the ALU.

```
┌─ ReadFromRAM ─────────────────────
│ Offset
├───────────────────────────────────
│ ¬((val dsf = 7) ∨ (val dsf = 6))
│  (val fsf ≠ 2) ∨ (val csf = 1)
│  io = 0
└───────────────────────────────────
```

Read in input to ALU from RAM. There is no read when a write is specified (ie if the dsf is 6 or 7), there is also no read from RAM when there is an input from PERI space (ie if the fsf is two and the csf is zero), finally the address of the location to be read from must be in the RAM space.

```
┌─ ReadFromPERI ────────────────────
│ Offset
├───────────────────────────────────
│ ¬((val dsf = 7) ∨ (val dsf = 6))
│ val csf = 0
│ val fsf = 2
│ io = 1
└───────────────────────────────────
```

Read in an input from the PERIpheral space.

$$Input \triangleq ReadFromRAM \lor ReadFromPERI$$

```
┌─NilMemoryRead──────────────────────
│ Offset
├───────────────────────────────────
│  val fsf = 12
│  val csf = 0
│ ¬((val dsf = 6) ∨ (val dsf = 7))
└───────────────────────────────────
```

This is the case where there is to be no word read in from memory, ie when the ALU function is a shift operation.

```
┌─MemoryRead──────────────────────────────
│ Offset
├────────────────────────────────────────
│ (val msf = 0)∧(m = offs) ∨
│ (val msf ≠ 0)∧(m = Mem ( offs trim 20 ))
└────────────────────────────────────────
```

This is the case where the memory read is to go ahead if msf is 0 then it is immediate addressing. otherwise get the value from the location pointed to by offs.

$$MemRead \triangleq (NilMemoryRead \lor \neg\, NilMemoryRead \land MemoryRead) \land Input$$

MemRead is either a nil memory read or a memory read.

4.2 Write to memory

```
┌─Write ──────────────────────────
│ ΔViper
├─────────────────────────────────
│ val csf = 0
│ (val dsf = 6) ∨ (val dsf = 7)
└─────────────────────────────────
```

Is the viper doing a write operation to main memory or peripheral space.

```
┌─Output ─────────────────────
│ Offset
├─────────────────────────────
│ (val dsf = 6)∧(io = 1) ∨
│ (val dsf = 7)∧(io = 0)
└─────────────────────────────
```

Define the region of memory where the write is to take place, ie either peripheral or main memory.

```
┌─Memwrite ─────────────────────────────────────
│ Offset
│ RegisterSelect
│ Write
│ Output
│ ΞRegisters
├───────────────────────────────────────────────
│ (val msf = 0)∧(δMem = {}) ∨
│ (val msf ≠ 0)∧(δMem = { (offs trim 20) ↦ r })
└───────────────────────────────────────────────
```

Write to main memory or peripheral space.

4.3 Illegal Operations

Illegal operations, which will cause an error.

```
│ invalid : Word → Bit
├──────────────────────────────────────────────────
│ ∀ w : Word •
│   (invalid w = 1) ⇔ (val w > maxval (wrd 20 0))
```

Function set true if the word cannot be held in a 20 bit word.

```
┌─SpareFunction ───────────────────────────────────
│ ΔViper
├──────────────────────────────────────────────────
│  val csf = 0
│ ¬( (val dsf = 6) ∨ (val dsf = 7) )
│  (val fsf = 13) ∨ (val fsf = 14) ∨ (val fsf =15)
└──────────────────────────────────────────────────
```

The Op code is accessing one of the three spare functions of the Vipers ALU.

```
┌─IllegalCall ─────────────────────────────────
│ ΔViper
├──────────────────────────────────────────────
│ val csf = 0
│ val fsf = 1
│ (val dsf = 0) ∨ (val dsf =1) ∨ (val dsf = 2)
└──────────────────────────────────────────────
```

The ALU operation is a Call, but the destination for the result is set to A, X or Y.

```
┌─IllegalPDestination ─────────────────────────────────────
│ ΔViper
├──────────────────────────────────────────────────────────
│ val csf = 0
│ (val dsf = 3) ∨ (val dsf = 4) ∨ (val dsf = 5)
│ ¬((val fsf = 1)∨(val fsf = 3)∨(val fsf = 5)∨(val fsf = 7))
└──────────────────────────────────────────────────────────
```

The destination for the result from the ALU is the Program counter. However the ALU function is an illegal way of generating the new Program Counter value.

```
┌─IllegalWrite ─
│ Write
├─────────────
│ val msf = 0
└─────────────
```

The operation is a write, but immediate addressing has been specified.

```
┌─ IllegalAddress ──────────────────────────────────────────────
│ Offset
├───────────────────────────────────────────────────────────────
│ (val csf = 1)∨(val fsf ≠ 12)∨(val dsf = 6)∨(val dsf = 7)
│ invalid offs
└───────────────────────────────────────────────────────────────
```

A memory location needs to be read, but the location to read from is not a valid address.

```
┌─ IllegalPIncrement ─
│ ΔViper
├─────────────────────
│ invalid newp
└─────────────────────
```

The Program Counter is to be incremented past the end of the address space.

```
┌─ Error ─────
│ ΔStop
│ ΞViper
├─────────────
│ sval = 1
│ P' = newp
└─────────────
```

Machine must stop with all registers are as they were previously.

```
Errors ≙ ((IllegalAddress  ∨  IllegalCall ∨  IllegalWrite
              ∨  IllegalPIncrement ∨  SpareFunction
              ∨  IllegalPDestination)
           ∧  Error)
```

4.4 Comparison Functions

```
┌─ CompareFrame ───
│ RegisterSelect
│ MemRead
├──────────────
│ P' = newp
│ A' = A
│ X' = X
│ Y' = Y
│ δMem = {}
│ val csf = 1
└──────────────
```

This is the framing schema for comparison operations. All registers are unchanged except for the Program counter. B' is set in the various comparisons below.

```
┌─ LessThan ──────────
│ CompareFrame
├──────────────────
│ val fsf =  0
│ val B' = (r less m)
└──────────────────
```

```
┌─ GreaterThanOrEqualTo ───
│ CompareFrame
├──────────────────────
│ val fsf =  1
│ val B' = not (r less m)
└──────────────────────
```

```
┌─ EqualTo ───────────
│ CompareFrame
├──────────────────
│ val fsf =  2
│ val B' = (r equal m)
└──────────────────
```

```
┌─ NotEqualTo ──────────
│ CompareFrame
├──────────────────────
│ val fsf =  3
│ val B' = not (r equal m)
└──────────────────────
```

```
LessThanOrEqualTo
CompareFrame
----------------------------------
val fsf =  4
val B' = (r less m) + (r equal m)
```

```
GreaterThan
CompareFrame
----------------------------------
val fsf =  5
val B' = not((r less m) + (r equal m))
```

```
UnsignedLessThan
CompareFrame
----------------------------------
val fsf =  6
val B' = (r borrow m)
```

```
UnsignedGreaterThanOrEqualTo
CompareFrame
----------------------------------
val fsf =  7
val B' = not(r borrow m)
```

```
LessThanOrB
CompareFrame
----------------------------------
val fsf =  8
val B' = (r less m) + val B
```

```
GreaterThanOrEqualToOrB
CompareFrame
----------------------------------
val fsf =  9
val B' = not(r less m) + val B
```

```
EqualToOrB
CompareFrame
----------------------------------
val fsf = 10
val B' = (r equal m) + val B
```

```
┌─ NotEqualToOrB ──────────────────────
│ CompareFrame
├──────────────────────────────────────
│ val fsf = 11
│ val B' = not(r equal m) + val B
└──────────────────────────────────────
```

```
┌─ LessThanOrEqualToOrB ─────────────────────────
│ CompareFrame
├────────────────────────────────────────────────
│ val fsf = 12
│ val B' = ((r less m) + (r equal m)) + val B
└────────────────────────────────────────────────
```

```
┌─ GreaterThanOrB ──────────────────────────────────
│ CompareFrame
├───────────────────────────────────────────────────
│ val fsf = 13
│ val B' =not( (r less m) + (r equal m) + val B)
└───────────────────────────────────────────────────
```

```
┌─ UnsignedLessThanOrB ────────────────
│ CompareFrame
├──────────────────────────────────────
│ val fsf = 14
│ val B' = (r borrow m) + val B
└──────────────────────────────────────
```

```
┌─ UnsignedGreaterThanOrEqualToOrB ────
│ CompareFrame
├──────────────────────────────────────
│ val fsf = 15
│ val B' = not (r borrow m) + val B
└──────────────────────────────────────
```

```
Compare ≙ UnsignedGreaterThanOrEqualToOrB ∨ LessThanOrB
          ∨ UnsignedLessThanOrB ∨ GreaterThanOrEqualToOrB
          ∨ GreaterThanOrB ∨ EqualToOrB ∨ UnsignedLessThan
          ∨ LessThanOrEqualToOrB ∨ NotEqualToOrB ∨ GreaterThan
          ∨ UnsignedGreaterThanOrEqualTo ∨ LessThanOrEqualTo
          ∨ NotEqualTo ∨ EqualTo ∨ GreaterThanOrEqualTo
          ∨ LessThan
```

Compare is the disjunction of all of the basic comparison schemas.

4.5 ALU Operations

```
ALUframe
  RegisterSelect
  MemRead
  Pwrite : Bit
  ----------------------------------------------------
  (Pwrite=1) ↔ ((val dsf=3)∨(val dsf=4)∨(val dsf=5))
  val csf = 0
  ¬((val dsf = 7) ∨ (val dsf = 6))
  δMem = {}
```

This is the framing schema for all of the ALU operations. Note memory cannot be changed and it is not a comparison. Pwrite is 1 if the destination of the result is the Program counter.

```
Negate
  ALUframe
  ----------------
  val fsf = 0
  result = wnot m
  B' = B
  sval = 0
```

Invert the input word.

```
Call
  ALUframe
  ----------------
  val fsf = 1
  result = m
  P' = m
  A' = A
  X' = X
  Y' = newp
  B' = B
  sval = not(Pwrite) + (invalid m)
```

Call a subroutine. Set Program counter to m, and leave the return address in the Y register. Stop if there is a call to an illegal address or there is not a legal P destination.

```
InputFromPERI
  ALUframe
  ----------------
  val fsf = 2
  result = m
  B' = B
  sval = 0
```

Input a value from the PERIpheral space. Note io has already been set to 1 in section 4.1 (ReadFromPERI).

```
┌─ReadFromMemory ──────────
│ ALUframe
├──────────────────────────
│ val fsf = 3
│ result = m
│ B' = B
│ sval=Pwrite · (invalid m)
└──────────────────────────
```

Return the value in memory, and stop if the location is not in memory space.

```
ReadOp ≙ (InputFromPERI ∨ ReadFromMemory )
```

The two read operations, ie the ALU is transparent.

```
┌─UnsignedAdd ─────────
│ ALUframe
├──────────────────────
│ val fsf = 4
│ result = r plus m
│ val B' = r carry m
│ sval = 0
└──────────────────────
```

Add r to m, setting B if there is a Carry.

```
┌─AddStopOnOverflow ──────────────────────────────────────
│ ALUframe
├─────────────────────────────────────────────────────────
│ val fsf = 5
│ result = r plus m
│ B' = B
│ sval = (r overflow m) + (invalid(result) · Pwrite)
└─────────────────────────────────────────────────────────
```

Add r to m, stopping if there is an overflow, and setting B if there is a Carry.

```
┌─UnsignedSubtract ─────
│ ALUframe
├───────────────────────
│ val fsf = 6
│ result = r minus m
│ val B' = r borrow m
│ sval = 0
└───────────────────────
```

Subtract m from r, and setting B if there is a Borrow.

```
┌─SubtractStopOnOverflow ──────────────────────────────
│ ALUframe
├──────────────────────────────────────────────────────
│ val fsf = 7
│ result = r minus m
│ B' = B
│ sval = (r underflow m) + (invalid(result) . Pwrite)
└──────────────────────────────────────────────────────
```

Subtract m from r, stopping on overflow and setting B if there is a Borrow.

```
ArithmeticOp  ≙ ( UnsignedAdd ∨ AddStopOnOverflow
                  ∨ UnsignedSubtract  ∨ SubtractStopOnOverflow )
```

The four arithmetic operations.

```
┌─ExclusiveOr ──────────
│ ALUframe
├───────────────────────
│ val fsf = 8
│ result = r exor m
│ B' = B
│ sval = 0
└───────────────────────
```

Returns the Exclusive Or of the two input words.

```
┌─And ──────────────────
│ ALUframe
├───────────────────────
│ val fsf = 9
│ result = r and m
│ B' = B
│ sval = 0
└───────────────────────
```

Returns the Logical and of the two inputs.

```
┌─Nor ──────────────────────
│ ALUframe
├───────────────────────────
│ val fsf = 10
│ result = wnot( r or m)
│ B' = B
│ sval = 0
└───────────────────────────
```

Returns the inverted or of the two inputs.

```
AndNot
ALUframe
----------------------------
val fsf = 11
result = r and wnot (m)
B' = B
sval = 0
```

Returns the logical and of the input register and the inverted memory input.

LogicalOp ≙ (Negate ∨ AndNot ∨ Nor ∨ And ∨ ExclusiveOr)

The five logical operators.

```
ArithmeticShiftRight
ALUframe
----------------------------
val fsf = 12
val msf = 0
result = MSB r >> r
B' = B
sval = 0
```

Arithmetic Shift Right, shifting in the MSB (ie sign) bit.

```
LogicalShiftRight
ALUframe
----------------------------
val fsf = 12
val msf = 1
result = val B >> r
val B' = LSB r
sval = 0
```

Logical Shift Right through the Boolean Flag B.

```
ArithmeticShiftLeft
ALUframe
----------------------------
val fsf = 12
val msf = 2
result = r plus r
B' = B
sval = (r overflow r)
```

Arithmetic Shift Left, Stopping the processor on overflow.

```
┌─LogicalShiftLeft ────
│ ALUframe
├──────────────────────
│ val fsf = 12
│ val msf = 3
│ result =  r << val B
│ val B' = MSB r
│ sval = 0
└──────────────────────
```

Logical Shift Left through the Boolean Flag B.

```
ShiftOp  ≙ (ArithmeticShiftRight ∨ LogicalShiftRight
           ∨ ArithmeticShiftLeft  ∨ LogicalShiftLeft )
```

The four shift operations.

```
┌─UnusedFunctions ──────────────────────────────────
│ ALUframe
├───────────────────────────────────────────────────
│ (val fsf = 13) ∨ (val fsf = 14) ∨ (val fsf = 15)
│ result =  r
│ B' = B
│ sval = 1
└───────────────────────────────────────────────────
```

The function called is one of the three unused functions in the Viper ALU. This will cause the Viper to stop.

```
ALU  ≙ ( ReadOp ∨ ArithmeticOp
       ∨ LogicalOp  ∨ ShiftOp ∨ UnusedFunctions )
```

The result from the ALU, is one of the above functional groups.

```
┌─ResultToA ──
│ ALUframe
├─────────────
│ val dsf = 0
│ A' = result
│ X' = X
│ Y' = Y
│ P' = newp
└─────────────
```

Store the result from the ALU in the A register.

```
┌─ResultToX ───
│ ALUframe
├──────────────
│ val dsf = 1
│ A' = A
│ X' = result
│ Y' = Y
│ P' = newp
└──────────────
```

Store the result from the ALU in the X register.

```
┌─ResultToY ───
│ ALUframe
├──────────────
│ val dsf = 2
│ A' = A
│ X' = X
│ Y' = result
│ P' = newp
└──────────────
```

Store the result from the ALU in the Y register.

```
┌─Jump ────────
│ ALUframe
├──────────────
│ val fsf ≠ 1
│ A' = A
│ X' = X
│ Y' = Y
│ P' = result
└──────────────
```

Branch instruction (as opposed to a call) simply set the program counter to be equal to the result from the ALU.

```
┌─Conditions ──────────────────────────
│ ALU
├──────────────────────────────────────
│ ( val dsf = 3 ) ∨
│ ( (val dsf = 4) ∧ (val B = 1) ) ∨
│ ( (val dsf = 5) ∧ (val B = 0) )
└──────────────────────────────────────
```

The values of dsf for the various conditional jumps and calls. For the unconditional call dsf is 3, call on B set dsf is 4 and call on B = 0 is dsf equal to 5.

```
Destination ≙ ( ResultToA ∨ ResultToX ∨ ResultToY  ∨
                  (Jump ∧ Conditions))

ALUOp ≙ (ALU ∧ Destination) ∨ (Call ∧  Conditions)
```

NoOp

ΞViper

---

val csf = 0

(val dsf = 5) $\wedge$ (val B = 1) $\vee$ (val dsf = 4) $\wedge$ (val B = 0)

P' = newp

4.6 Next Viper State

$$OKState \mathrel{\widehat{=}} \neg (Errors) \land (Memwrite \lor NoOp \lor Compare \lor ALUOp \lor Reset)$$

$$NextState \mathrel{\widehat{=}} (Errors \lor OKState \lor Stopped \lor Reset)$$

The next state is one of four cases, it is either stopped or an error in which case the next state will be stopped, or it will be a reset and the next state will be the initial state, or it will continue to work normally.

5 Conclusions

The document gives an initial specification of Viper1 in Z. It has demonstrated that Z can give a higher level specification than the HOL specification. Z has also been shown to be a useful language to specify a microprocessor in.

Although this specification has been written some time after the HOL specification, it was still a worthwhile exercise. This specification can be checked against the HOL version. The experience gained has also been useful in specifying Viper2.

6 Acknowlegements

W J Cullyer, C Pygott and J Kershaw for their help with the *Viper1* and the HOL spec.

C O'Halleran for his help with the Z editor and type checker.

S Wiseman for checking the specification, and suggesting modifications.

7 References

1. Bowen J. *The Formal Specification of a Microprocessor* instruction set.

2. *Cullyer W.J* Viper Microprocessor:Formal Specification RSRE Report No. 85013 October 1985.

3. Ian Hayes (editor) Specification Case Studies Prentice-Hall *international series in* computer science, 1987

DOCUMENT CONTROL SHEET

Overall security classification of sheet ...UNCLASSIFIED.. ....

(As far as possible this sheet should contain only unclassified information. If it is necessary to enter classified information, the box concerned must be marked to indicate the classification eg (R) (C) or (S))

1. DRIC Reference (if known)	2. Originator's Reference Memorandum 4195	3. Agency Reference	4. Report Security Classification Unclassified

5. Originator's Code (if known) 7784000	6. Originator (Corporate Author) Name and Location Royal Signals and Radar Establishment St Andrews Road, Malvern, Worcestershire WR14 3PS
5a. Sponsoring Agency's Code (if known)	6a. Sponsoring Agency (Contract Authority) Name and Location

7. Title

SPECIFICATION OF VIPER1 IN Z

7a. Title in Foreign Language (in the case of translations)

7b. Presented at (for conference papers) Title, place and date of conference

8. Author 1 Surname, initials Kemp D H	9(a) Author 2	9(b) Authors 3,4...	10. Date 1988.9	pp. ref. 34

11. Contract Number	12. Period	13. Project	14. Other Reference

15. Distribution statement

Unlimited

Descriptors (or keywords)

continue on separate piece of paper

Abstract

The Viper1 microprocessor has already been specified mathematically in HOL. HOL, however, is not well known outside the hardware verification community. This paper covers the specification Viper1 in the Z specification language. Various features of Viper1 have been specified in Z which did not occur in the top level HOL specification. It has not been possible to prove any correspondence between this specification and the original HOL specification. The work involved in writing the Viper1 specification has proved useful in writing the initial Viper2 specification.